Information, Calcul et Communication
Théorie du Calcul

Pr. B. Moret & J.-C. Chappelier

Objectif du cours d'aujourd'hui

Revenir sur la définition théorique de « calcul » / algorithme

Etudier les deux grandes questions de la théorie du calcul :

- que peut-on résoudre avec un algorithme ?
- que peut-on résoudre *efficacement* avec un algorithme ?

Calcul ?

Dans la leçon I.1 nous avons défini un algorithme comme : la composition d'un ensemble fini d'*opérations élémentaires bien définies* (déterministes) opérant sur un nombre fini de données et effectuant un traitement bien défini.

Mais quelles sont, formellement, ces *opérations élémentaires* ?

Il y a en fait plusieurs définitions (formelles) de la notion de calcul, parmi lesquelles les plus connues sont :

- le λ-calcul (Church)
- les fonctions récursives (Gödel, Church, Kleene)
- les machines de Turing (Turing)

Ce qui est « incroyable », c'est que toutes ces définitions sont *équivalentes* : notion de *Turing-équivalent* (ou strictement Turing-complet)

Voyons ici la plus intuitive : les machines de Turing

Machines de Turing : définition (1/2)

Une machine de Turing est un automate abstrait (= objet *mathématique*), constitué des éléments suivants :

- une **bande** infinie (c.-à-d. de taille non bornée), décomposée en cellules au sein desquelles peuvent être « stockés » des **caractères** (issus d'un ensemble fini Σ appelé *alphabet*).
- une **tête de lecture/écriture**, pouvant :
  - lire et modifier le caractère stocké dans la cellule correspondant à la position courante de la tête (le caractère courant)
  - se déplacer d'une cellule vers la gauche ou vers la droite (c.-à-d. modifier sa position)
- un ensemble fini E d'**états internes** servant à caractériser le fonctionnement de la machine.
  On définit aussi un état initial et des états finaux.
- une **table de transitions**

Machines de Turing : définition (2/2)

Une machine de Turing est un automate abstrait (= objet *mathématique*), constitué des éléments suivants :

- une **bande** infinie contenant des **caractères**
- une **tête de lecture/écriture**
- un ensemble fini E d'**états internes**
- une **table de transitions** indiquant, pour chaque couple
  (état interne, caractère courant)
  une nouvelle valeur pour ce couple, ainsi que le déplacement de la tête de lecture/écriture.

  Dans la table de transitions, chaque couple est donc associé à un triplet :
  (nouvel état interne, nouveau caractère, déplacement)

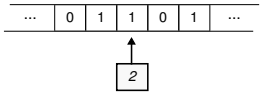
  C'est donc une application (au sens mathématique) de $E \times (\Sigma \cup \{\varepsilon\})$ dans $E \times (\Sigma \cup \{\varepsilon\}) \times \{-,+\}$

  (Note : ε représente ici le caractère vide, « pas de caractère ».)

Machines de Turing : exemple

$\Sigma = \{0,1\}$

$E = \{1,2,3\}$ (1 : état initial, 3 : état final)



état courant \ caractère courant	0	1	ε
1	(1,0,+)	(1,0,+)	(2,ε,-)
2	(2,0,-)	(2,0,-)	(3,ε,+)

avec + (respect. -) indiquant un déplacement vers la droite (respect. vers la gauche).

Machines de Turing : fonctionnement

Une machine de Turing fonctionne alors selon le principe suivant :

1. la bande est initialisée avec la séquence de caractères correspondant aux (/à un codage des) données d'entrée ;
2. la tête de lecture/écriture est positionnée sur la première cellule de l'entrée (= la plus à gauche), et l'état interne est positionné à sa valeur initiale (qui fait partie de la définition de la machine).
3. tant que l'état interne courant n'est pas un état final (ce qui fait partie de la définition de la machine), le triplet
   (état interne[nouveau], caractère[nouveau], déplacement)
   est utilisé pour mettre à jour l'état interne, le caractère courant, puis le déplacement de la tête de lecture/écriture est effectué.
4. si l'état interne courant est un état final, la machine s'arrête.

   Le contenu de la bande à ce moment-là est considéré comme le résultat du traitement.

Exemple : déterminer si un nombre est pair

Entrée : le nombre à tester, écrit en binaire
Sortie : 1 si le nombre est pair, 0 sinon

	0	1	ε
1	(1,0,+)	(1,1,+)	(2,ε,–)
2	(3,ε,-)	(4,ε,-)	(inutile)
3	(3,ε,-)	(3,ε,-)	(5,1,+)
4	(4,ε,-)	(4,ε,-)	(5,0,+)
5	(inutile)	(inutile)	(6,ε,-)

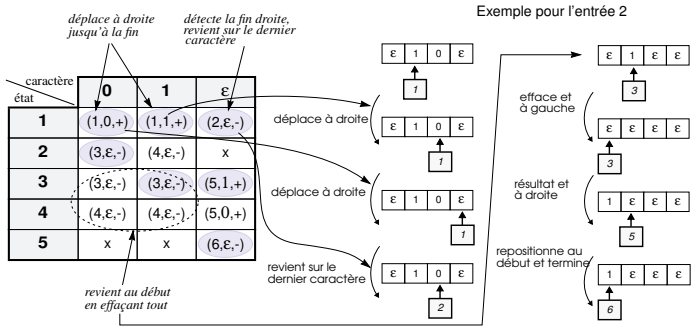
Exemple : entrée :

$\cdots \varepsilon\ 1\ 0\ \varepsilon \cdots$

Résumé de l'exemple de fonctionnement

Machine de Turing déterminant si un nombre est pair

Entrée : le nombre à tester, sous forme binaire
Sortie : 1 si le nombre est pair, 0 sinon



état \ caractère	0	1	ε
1	(1,0,+)	(1,1,+)	(2,ε,-)
2	(3,ε,-)	(4,ε,-)	x
3	(3,ε,-)	(3,ε,-)	(5,1,+)
4	(4,ε,-)	(4,ε,-)	(5,0,+)
5	x	x	(6,ε,-)

Traduction de l'exemple de fonctionnement

	0	1	ε
1	(1,0,+)	(1,1,+)	(2,ε,–)
2	(3,ε,-)	(4,ε,-)	(inutile)
3	(3,ε,-)	(3,ε,-)	(5,1,+)
4	(4,ε,-)	(4,ε,-)	(5,0,+)
5	(inutile)	(inutile)	(6,ε,-)

Tant que lu = 0 ou lu = 1
  Avance (vers la droite)
Recule (vers la gauche) de 1
Si lu = 0
  Tant que lu = 0 ou lu = 1
    Recule en effaçant
  Écrit « 1 »
Sinon
  Tant que lu = 0 ou lu = 1
    Recule en effaçant
  Écrit « 0 »
Recule de 1

(Remarque : forme canonique)

On voit qu'une machine de Turing est caractérisée par

1. sa logique générale de fonctionnement (c.-à-d. la définition) ;
2. le codage de ses entrées et sorties (par exemple sous forme de séquences binaires) ;
3. la table de transitions décrivant son fonctionnement.

Si l'on impose de coder les entrées et les sorties en binaire (et d'indiquer l'absence de caractère dans une cellule par le caractère ε), on obtient une représentation uniforme des machines de Turing, appelée **représentation canonique**.

Remarque :
D'autres choix sont possibles pour la définition des machines de Turing (plusieurs bandes, autres opérations élémentaires, autres alphabets de caractères, ...) mais on peut montrer que les résultats théoriques obtenus avec de telles machines sont équivalents à ceux obtenus à l'aide d'une machine de Turing canonique.

Machine de Turing universelle (1)

Le fonctionnement d'une machine de Turing est conditionné par sa table de transitions.

☞ une machine de Turing = abstraction d'un automate de comportement non modifiable *a posteriori*...
(c.-à-d. **1 machine de Turing donnée = 1 algorithme *donné***)

...ce n'est **pas** encore un automate *programmable*, c.-à-d. une machine pour laquelle **le programme fait partie des données d'entrée**
(et non être un élément constitutif de la machine (table de transitions))

Si l'on désire qu'une machine de Turing constitue une abstraction pour la notion d'automate programmable, il faut que sa table de transitions soit fixe, et que le conditionnement de son fonctionnement soit entièrement imposé par ses données d'entrées.

Machine de Turing universelle (2)

c.-à-d. que bien qu'ayant une table de transitions fixe, elle puisse *effectuer des tâches différentes* décrites uniquement par ses données d'entrée
(ces données d'entrée sont donc à la fois le programme et les données au sens usuel)

Une telle machine s'appelle **LA machine de Turing universelle**

C'est une machine de Turing permettant de simuler le fonctionnement de n'importe quelle autre machine de Turing, tout en ayant une table de transitions fixe.

Mais comment la construire ?

Est-ce possible ?

Construction de la machine de Turing universelle

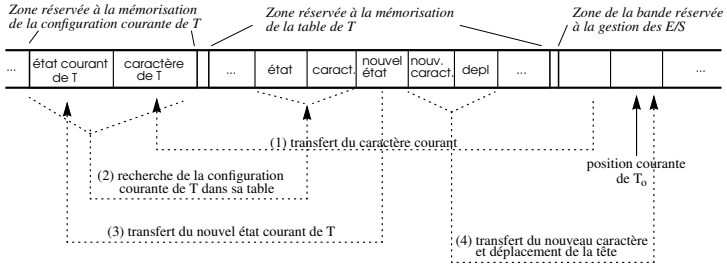
On peut montrer que l'on peut effectivement construire une machine de Turing universelle
(mais cela dépasse le cadre de ce cours)

L'idée permettant de construire une telle machine est la suivante :

1. la table de transitions de la machine T à simuler est codée sous la forme d'une séquence binaire ;
2. la bande de la machine universelle est séparée en deux zones distinctes, l'une permettant de stocker la séquence binaire représentant la table de la machine T, ainsi que sa configuration (état interne, caractère courant), et l'autre permettant de gérer les données d'entrées (et sortie) de T.

Fonctionnement de la machine de Turing universelle

Le fonctionnement de la machine universelle peut alors être schématisé ainsi :



Intérêt des machines de Turing

Les machines de Turing constituent une notion centrale en informatique, car elles permettent de donner :

1. une définition précise à la notion informelle **d'algorithme**
   (1 algorithme = 1 table de transitions d'*une* machine de Turing)

2. une base théorique solide aux notions importantes que sont la **calculabilité**, la **décidabilité** et la **complexité**.

Une question et des données

En leçons 1 et 2 nous avons vu des algorithmes pour résoudre des problèmes de **recherche**, **tri**, et **plus courts chemins**.

Ces algorithmes fonctionnent *quelles que soient les données*.

Peu importent la taille des données, le nom de la personne recherchée, le type d'objets à trier, les valeurs des distances entre deux points, etc.

Chaque algorithme résout chaque fois *la même question*.

Est-ce qu'une certaine personne est présente dans la salle ? De quelle manière devons-nous arranger une collection d'objets pour qu'ils soient en ordre croissant ? Quel est le chemin le plus court entre deux points ?

Faire tourner l'algorithme sur une entrée nous donne
*la réponse à une **instance** du problème*

Faire tourner un algorithme n'est pas la solution du problème :
la solution, c'est l'algorithme lui-même

Exemple de problème : Prem

entrée : n entier naturel (≥ 2)

Question (= sortie voulue d'algorithme solution de ce problème) :
n est-il premier ?

Instances :
2 est-il premier ?
3 est-il premier ?
4 est-il premier ?
...

Solution : algorithme résolvant *n'importe quelle* instance

par exemple :
1. si n est pair, il n'est premier que s'il vaut 2
2. sinon, tester la division de n par tous les entiers impairs plus petits que $\sqrt{n}$

Problèmes et problèmes intéressants

Définitions :
Un problème se compose d'une question et d'un ensemble d'instances.
Une solution pour un problème est un algorithme qui répond correctement à la question *pour chaque instance* possible.

Observation :
Si l'ensemble d'instances est fini, il est toujours possible de résoudre le problème en bâtissant une table de correspondances qui stocke la réponse pour chaque instance.
L'« algorithme » de résolution consulte cette table et imprime la solution y trouvée — il n'y a pas de calcul ni d'algorithme !

Conclusion :
Seuls les problèmes avec des ensembles **infinis** d'instances demandent du travail : ce sont les seuls qui soient intéressants.

(pour les théoriciens)

Apprendre à compter

Nous savons bien sûr tous compter : $1,2,3,4,5,\ldots$
Mais comment compter un ensemble infini ?

Observation :
Les entiers positifs, $\mathbb{N}^* = 1,2,3,\ldots$, définissent le comptage.

Définition : un ensemble S est dénombrable si et seulement si il existe une surjection $f : \mathbb{N}^* \to S$.
La fonction f fait le numérotage de S.

Exemple : *Les entiers, $\mathbb{Z} = \ldots,-3,-2,-1,0,1,2,3,\ldots$, forment un ensemble dénombrable.*
On peut prendre la fonction $f : \mathbb{N} \to \mathbb{Z}$ comme suit :

si i est pair, alors $f(i) = i/2$
si i est impair, alors $f(i) = -(i-1)/2$

Le numérotage est le suivant (numéro en bleu, valeur en rouge) :
1 :0, 2 :1, 3 :-1, 4 :2, 5 :-2, 6 :3, 7 :-3, . . .

Les paires d'entiers positifs sont dénombrables

« *Sûrement pas ! Après tout, il y a un nombre infini de paires qui ont toutes le même premier élément !* »

Et pourtant. . .

Ecrivons les paires sur un tableau comme ci-dessous :

1,1	1,2	1,3	1,4	1,5	. . .
2,1	2,2	2,3	2,4	2,5	. . .
3,1	3,2	3,3	3,4	3,5	. . .
4,1	4,2	4,3	4,4	4,5	. . .
5,1	5,2	5,3	5,4	5,5	. . .

Énumérons les paires sur la base de leur somme.

Nous commençons donc par $(1,1)$, puis $(1,2)$ et $(2,1)$, puis $(1,3)$, $(2,2)$ et $(3,1)$, puis $(1,4)$, $(2,3)$, $(3,2)$ et $(4,1)$, etc.

Chaque paire reçoit son numéro unique—nous avons défini une surjection.

(Note : cela prouve également que $\mathbb{Q}$ est dénombrable.)

Les algorithmes sont dénombrables

Un algorithme est simplement un texte écrit à l'aide d'un alphabet choisi (p.ex. table d'une machine de Turing). Il est facile d'énumérer (numéroter) tous les textes possibles dans un ordre lexicographique :

- Commençons par énumérer les textes d'un seul caractère :
  1 :a, 2 :b, 3 :c, . . ., 26 :z
- Continuons avec les textes de deux caractères :
  27 :aa, 28 :ab, 29 :ac, . . ., 52 :az, 53 :ba, 54 :bb, . . .,
  677 :za, 678 :zb, 679 :zc, . . ., 702 :zz
- Maintenant passons aux textes de trois caractères :
  703 :aaa, 704 :aab, 705 :aac, . . ., 18'278 :zzz,
  et ainsi de suite.

Même avec le bon choix d'alphabet, certains des textes énumérés ne sont pas des algorithmes, mais tous les algorithmes sont énumérés.

Les fonctions booléennes ne sont pas dénombrables

Soit l'ensemble des fonctions booléennes d'une variable entière

$$B = \{f \mid f\colon \mathbb{N}^* \to 0,1\}$$

Chaque fonction peut être représentée par une liste de valeurs binaires :

n	1	2	3	4	5	6	7	. . .
$f(n)$	0	0	0	1	0	0	1	. . .

Si l'ensemble B était dénombrable, nous pourrions numéroter les fonctions : f_1, f_2, f_3, f_4, f_5, f_6, f_7, . . .

Ecrivons-les en tableau : Observons la diagonale : une liste de valeurs binaires—une fonction $f \in B$

f_1	0	1	1	0	0	0	1	. . .
f_2	0	0	0	0	1	0	1	. . .
f_3	1	1	0	0	1	1	1	. . .
f_4	0	1	0	1	0	1	1	. . .
f_5	1	0	1	0	0	0	0	. . .
f_6	1	1	0	1	1	0	1	. . .
f_7	0	0	0	1	1	1	1	. . .

Les fonctions booléennes ne sont pas dénombrables

Si l'ensemble B était dénombrable, nous pourrions numéroter les fonctions :

f_1, f_2, f_3, f_4, f_5, f_6, f_7, . . .

La diagonale est une liste de valeurs binaires—une fonction $f \in B$

f_1	0	1	1	0	0	0	1	. . .
f_2	0	0	0	0	1	0	1	. . .
f_3	1	1	0	0	1	1	1	. . .
f_4	0	1	0	1	0	1	1	. . .
f_5	1	0	1	0	0	0	0	. . .
f_6	1	1	0	1	1	0	1	. . .
f_7	0	0	0	1	1	1	1	. . .

$f : 0\,0\,0\,1\,0\,0\,1 \ldots$

Cette fonction peut être écrite $f(i) = f_i(i)$

(L'indice et la variable ont la même valeur sur la diagonale.)

Définissons une nouvelle fonction booléenne $f^*(i) = 1 - f_i(i)$

$f^* : 1\,1\,1\,0\,1\,1\,0 \ldots$

Les fonctions booléennes ne sont pas dénombrables

$$f^*(i) = 1 - f_i(i)$$

f^* **n'apparaît pas dans l'énumération !** *(pour chaque numéro possible i, nous avons $f^*(i) \neq f_i(i)$)*

$\Longrightarrow$ B n'est pas dénombrable. *(Pour tout numérotage, il y a toujours une fonction $f^* \in B$ non numérotée.)*

Pas assez d'algorithmes !

Qu'avons-nous appris ?

L'ensemble de tous les algorithmes (quel que soit le formalisme) est dénombrable.

L'ensemble des fonctions booléennes d'une variable entière n'est pas dénombrable.

Donc il existe *beaucoup plus* de fonctions booléennes que d'algorithmes.

Donc la plupart des fonctions booléennes *ne peuvent pas être calculées* par un algorithme !
(Il n'y a simplement pas assez d'algorithmes possibles.)

Deux paradoxes à 2500 ans d'écart

Pouvons-nous donner des exemples concrets de fonctions booléennes qui ne peuvent pas être calculées ?

Oui, avec l'aide de deux paradoxes fameux :

- *Le paradoxe d'Épiménide (ou paradoxe du menteur)*
- *Le paradoxe de Berry*

Épiménide fut (peut-être) un philosophe Crétois qui aurait vécu il y a plus de 2500 ans ; il aurait dit « tous les Crétois sont des menteurs ».
Ceci n'est pas un vrai paradoxe (si Épiménide mentait, il n'y a pas de contradiction) ; il aurait plutôt du dire « je suis en train de vous mentir ».
Ce « paradoxe » nous mène au problème de l'arrêt.

Berry fut bibliothécaire à Oxford à la fin du XIX^e siècle. Lui aussi avait mal exprimé son paradoxe ; cela fut corrigé par le célèbre mathématicien Bertrand Russell :
« soit n le plus petit entier positif qui ne peut pas être défini en moins de 20 mots » ;
un texte de 16 mots.
Ce paradoxe nous mène au problème de la longueur minimale de description.

Le problème de l'arrêt

Problème de l'arrêt : Existe-t-il un algorithme A qui, étant donné un autre algorithme P et des données x, décide si P, appliqué à x, s'arrête ou non ?

Théorème : Il n'existe pas d'algorithme pour résoudre le problème de l'arrêt.

La preuve est simple et se fait par l'absurde. Supposons qu'un tel algorithme A existe. Alors $A(P,x)$ sort « oui » si $P(x)$ s'arrête, et sort « non » sinon.
Notons que P, A, et x sont tous des textes et écrivons alors un nouvel algorithme I (Impossible) comme suit :

$I(x)$: Tant que $A(x,x)$;

Que se passe-t-il si nous utilisons le texte I en entrée de l'algorithme I ?

$I(I)$ fait tourner $A(I,I)$:
Si $A(I,I)$ répond « non »—ce qui veut dire que $I(I)$ ne s'arrête pas—,
alors $I(I)$ s'arrête : contradiction.
Si $A(I,I)$ répond « oui »—ce qui veut dire que $I(I)$ s'arrête—,
alors $I(I)$ entre dans une boucle infinie : contradiction.

La longueur minimale de description

Une question de base pour la compression des messages est simplement :
quelle est la taille minimale d'un message transmettant l'information désirée ?

On appelle « *complexité de Kolmogorov* » de n (notée ici $K(n)$), la longueur **minimale** d'un algorithme qui produise n.

Théorème : La complexité de Kolmogorov n'est pas calculable : il n'existe pas d'algorithme pour déterminer $K(n)$.

La preuve se fait par l'absurde. Supposons qu'un tel algorithme L existe : $L(n)$ retourne $K(n)$, la longueur de l'algorithme le plus court qui puisse écrire n. Définissons alors un nouvel algorithme B comme suit :

B : $i \longleftarrow 0$; Répéter $i \longleftarrow i+1$ tant que $L(i) \leq 60$; Sortir : i

B retourne donc le plus petit entier i tel que $K(i) > 60$, c.-à-d. tel qu'aucun algorithme de longueur moindre que 60 ne puisse écrire i.

Quelle est la longueur de B ? 57 caractères (espaces comprises).

B calcule le plus petit i tel qu'aucun algorithme de longueur moindre que 60 ne puisse écrire i, mais B lui-même écrit ce i et sa longueur est moindre que 60 : contradiction !

Quelques remarques sur $K()$

- Le fait que K ne soit pas calculable :
  - **ne** signifie **pas** : pour tout n, on ne peut pas calculer $K(n)$:
    par exemple, $K(1)$ est très certainement connu (10 en « français » : Sortir 1 ; ; et aussi une dizaine pour la (table de la) plus petite machine de Turing qui le fait)
  - mais veut dire : **il existe** des n, pour lesquels on ne peut pas calculer $K(n)$.
- Pourquoi $K(i)$ n'est il pas simplement $\log_2(i)$: écrire i en binaire ?
  1. Tout d'abord parce que la longueur de l'algorithme en question :
     Ecrire b_1 ; Ecrire b_2 ; ...
     ne serait pas $\log_2(i)$, mais $(1+U)\cdot\log_2(i)$ où U est la longueur de l'écriture de l'instruction « Ecrire » (8 en « français » avec les deux espaces et le point-virgule ; une dizaine caractères pour la partie de la table de la machine de Turing correspondante)
  2. Cela n'est certainement par la longueur **minimale** !
     Considérez par exemple : « Pour i de 1 à 10 : Ecrire 1 ; »
     (pour de tels nombres, $K(n)$ est donc plutôt de l'ordre de grandeur de $\log_2(\log_2(n))$)

Autres problèmes indécidables

- Un algorithme contient-il un morceau de code inutile ? (Th. de Rice)
- Deux algorithmes calculent-ils la même chose ? (Th. de Rice)
- Une configuration du jeu de la vie de Conway finit-elle par disparaitre ou persiste-t-elle toujours ?
- Peut-on paver le plan sans recouvrement ni espace vide avec un ensemble de formes géométriques ?

Quand un problème est indécidable, on **SAIT** qu'on ne peut pas déterminer de réponse à la question posée :

il est inutile de perdre du temps sur cette formulation d'un problème ou de croire quelqu'un qui voudrait nous vendre une solution à ce problème...

Cela est *beaucoup plus fort* que de dire qu'on ne sait pas le résoudre !

En résumé

La plus grande partie des problèmes que l'on peut définir par les mathématiques n'ont pas de solution algorithmique.

L'aspect clef de ces problèmes est l'autoréférence, un outil (presque trop) puissant en informatique.

La plupart de ces problèmes sont artificiels et n'ont aucun intérêt,
mais certains sont importants dans la pratique, tels que des questions de caractéristiques des algorithmes.

Cependant, tout n'est pas perdu : il peut exister des algorithmes qui résolvent les instances courantes de ces problèmes.

La complexité : que mesurer et comment ?

Certaines tâches informatiques doivent se faire des millions de fois par seconde dans le monde entier (transactions bancaires) ; d'autres demandent des milliards de milliards d'opérations (modèles climatiques) ; enfin d'autres se font sur des données gigantesques (Google, génome humain).
Toutes demandent donc des algorithmes aussi efficaces que possible.

Comment savoir si notre algorithme est le plus efficace possible ?

C'est une caractéristique du problème, pas de l'algorithme !

Il faut donc mesurer la complexité des problèmes.
On le fait pour le temps de calcul (temps) et pour les demandes de stockage (espace).

Pour que ces mesures soient robustes, il faut qu'elles soient applicables pour tout outil de calcul, du smartphone au supercomputer. Elles ne peuvent donc pas être très précises.
☞ $\Theta(\cdot)$

La complexité : une hiérarchie

La théorie du *calcul* partage les problèmes entre ceux qui **peuvent** et ceux qui *ne peuvent pas* être résolus par le calcul.

La théorie de la *complexité* n'étudie que les *problèmes* pour lesquels un algorithme de solution existe et partage ces problèmes en fonction des demandes de temps ou d'espace de l'algorithme **le plus efficace possible** pour chaque problème.

Ce partage se fait en grandes classes de complexité, chacune étant un ensemble de problèmes pour lesquels les ressources nécessaires (temps ou espace) à leur algorithme le plus efficace sont à peu près les mêmes.

Le résultat est une hiérarchie de classes de complexité.

Quelles classes devrions-nous définir ?

Algorithmes « pratiques »

Nous avons étudié dans les leçons précédentes des algorithmes de recherche, de tri, et de plus court chemin. Leurs temps de calcul variaient de logarithmique (dichotomie) à cubique (plus court chemin).

En général, un algorithme « pratique » prend au plus un temps de calcul cubique (rappel : la mesure est en fonction de la taille de l'entrée)—que l'on va généraliser à temps polynomial.

Parenthèse : Par contre, pour le volume de données de Google, du CERN, ou de la NASA, temps cubique est exclu : temps linéaire est requis.

Exemple : un ordinateur moderne fait de l'ordre de 10 milliards (10^{10}) d'opérations par seconde. Le CERN génère près d'un petaoctet (10^{15} octets) de données par semaine. Un algorithme quadratique ferait au moins 10^{30} opérations sur ces données, demandant de l'ordre de $10^{30}/10^{10} = 10^{20}$ secondes—ou 3 milliards de siècles !
Même un algorithme linéaire ferait au moins 10^{15} opérations, demandant 10^5 secondes, ou à peu près 30 heures.

Résoudre en temps polynomial : P

La classe de complexité devenue synonyme de solution efficace est *P*, la classe des problèmes pour lesquels il existe un algorithme qui résout n'importe quelle instance de ce problème en temps polynomial.

Vu que les ordinateurs modernes ne peuvent pas utiliser plus qu'un nombre constant d'octets de stockage à chaque instruction, temps polynomial implique espace polynomial.

Bien entendu, ces polynômes sont typiquement cubiques ou plus petits : personne n'a conçu d'algorithme sérieux en $\Theta(n^{17})$...

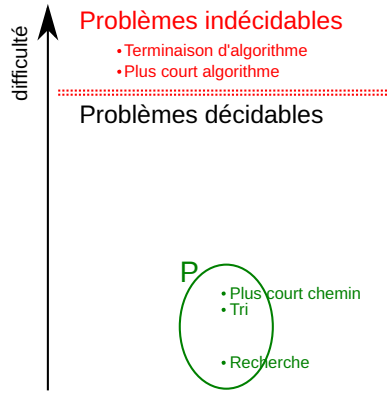
Exemples de problèmes dans P

Ouvrez n'importe quel livre sur les algorithmes : vous trouverez des centaines de pages décrivant des problèmes dans la classe P et leurs algorithmes de solution.

- la *recherche* par dichotomie : $\log n$
- (*recherche* linéaire) trouver tous les amis dans un réseau social : n
- l'intersection de deux polygones convexes : n
- le *tri* : $n \log n$
- le *chemin le plus court* dans un graphe : n à n^3 (en fonction des contraintes)
- l'alignement de deux séquences d'ADN : n^2
- l'affectation du personnel de bord aux vols de ligne : $n^2 \log n$
- la multiplication de deux matrices carrées the taille $n \times n$: $n^{2.37}$
- l'optimisation du débit entre deux points d'un réseau : n^3

(Note : n est la « taille de l'entrée » du problème : nombre d'éléments d'une liste, nombres de nœuds d'un graphe, ...)

Résumé (à ce stade) de la classification des problèmes



Oracles, certificats, et vérification

Au cours des ages, l'humanité a voulu des réponses de ses divinités favorites, souvent en échange pour des prières ou des sacrifices. Dans la période classique en Grèce et chez les Romains, il y avait pour cela faire des oracles.

Un oracle donne la bonne réponse et prend un temps de calcul constant.

Les oracles de Grèce étaient très concis—avec les résultats confondants que l'on sait.

En informatique, nous allons exiger qu'une réponse puisse être vérifiée avec l'aide d'un **certificat**.

Exemple :
Vous : « *Puis-je me rendre de Lausanne à Zurich par train en moins de 3h ?* »
Oracle : « *Oui.* »
Vous : « *Hmmm...* »

Il vous faut un certificat : quels trains prendre à quelle heure, du départ à l'arrivée.
Un tel certificat est facilement vérifié en temps linéaire avec l'aide d'un horaire officiel.

Certificat

Un **certificat** est une preuve (concise) qu'une instance est bien une instance positive du problème.

Exemple de certificat de « **non-**Prem » : un diviseur (et l'algorithme de division)

par exemple : 5 est un certificat que non-Prem(15) est une instance positive
(on vérifie *facilement* que 15 est divisible par 5)

Exemple de certificat de « Prem » : beaucoup plus compliqué...
(un nombre p et un algorithme *efficace* qui vérifient que $p^{n-1} = 1 \mod n$
et $p^{(n-1)/q} \neq 1 \mod n$ pour tout q facteur premier de $n-1$.)

par exemple : 5 est un certificat que Prem(2017) est une instance positive.

voir : test de primalité de Lucas-Lehmer ; certificat de Pratt ;
`https://fr.wikipedia.org/wiki/Test_de_primalit%C3%A9_de_Lucas-Lehmer`)

Vérifier en temps polynomial : NP

La classe de complexité caractérisant les problèmes dont les « solutions » (= instances positives) peuvent être vérifiées efficacement (en temps polynomial) avec l'aide d'un certificat est la classe *NP*.

Remarque : *La différence entre P et NP (tout deux définis en termes de temps polynomial) est celle entre résoudre (répondre sur toute instance) et vérifier (une instance positive donnée).*

*NP **contient** P — si l'on peut résoudre le problème soi-même en temps polynomial, on crée (et vérifie) son propre certificat en chemin.*

Mais résoudre est apparemment plus difficile que vérifier : nous connaissons des centaines de problèmes dans la classe NP pour lesquels nous n'avons pas pu trouver d'algorithme de solution qui tourne en temps polynomial.

Le plus grand problème théorique d'aujourd'hui (en informatique et en math) :
est-ce que *P* est égal à *NP* ?

P versus NP

ATTENTION ! NP **ne veut pas dire non-**polynomial mais
non-déterministe polynomial

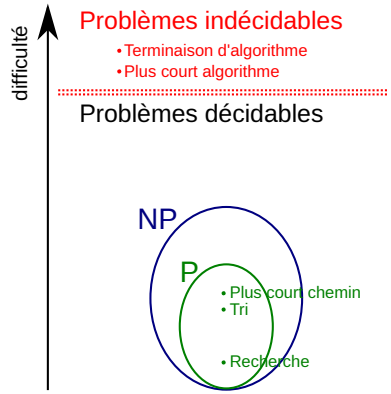
☞ La confusion est fréquente ! !

Le non-déterminisme réside dans le fait que l'on présuppose qu'une solution peut-être devinée par un moyen quelconque (oracle).

Si on sait le faire, et si on peut vérifier de façon efficace que la solution en est bien une, alors le problème est dans la classe NP.

☞ et (on répète) : P ⊂ NP

Résumé (d'une partie) de la classification des problèmes



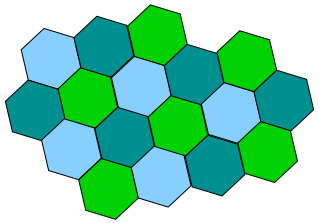
Positionner un problème : exemple des problèmes de coloration de carte

Soit une carte géographique à colorier de telle sorte que deux cantons voisins n'aient pas la même couleur :

Problème 1 : Peut-on la colorier avec **deux couleurs** ?

Problème 2 : Peut-on la colorier avec **trois couleurs** ?

Problème 3 : Peut-on la colorier avec **quatre couleurs** ?



Les problèmes de coloration de graphes (1/2)

Ces trois problèmes ont l'air de beaucoup se ressembler.

Et pour chacun d'eux, on sait proposer un algorithme de recherche exhaustive inefficace (exponentiel) :

1. Générer toutes les colorations possibles.
2. Regarder s'il en existe une satisfaisant le critère requis.

☞ pour 3 couleurs et 16 cantons, ça ne fait « que » $3^{16} = 43$ millions de coloriages possibles !

Mais, sait-on faire mieux ?

Les problèmes de coloration de graphes (2/2)

Sait-on faire mieux qu'un temps exponentiel pour répondre à ces problèmes ?

Oui pour le problème 1 : on sait y répondre rapidement (complexité linéaire)

☞ il suffit de vérifier qu'il n'existe aucun point de la carte où se rencontrent un nombre impair de cantons (polynomial)

Non, on ne sait pas faire mieux (à ce jour) pour le problème 2

☞ aucun algorithme polynomial n'est connu et ce problème est prouvé être dans NP (même NP-complet)

...et à nouveau **oui**, on peut répondre efficacement au problème 3
(et la réponse est toujours « oui » $\rightarrow \Theta(1)$)

☞ démontré pour toute carte par Appel et Haken, en 1976. [1] [2] [3]

1. C'était la première fois qu'on utilisait un ordinateur pour démontrer un théorème : était-ce bien encore une démonstration ? Cet algorithme a été amélioré en 1995 par Robertson, Sanders, Seymour et Thomas.
2. Ce théorème répond en $\Theta(1)$ au problème de *décision* ; il ne dit pas en $\Theta(1)$ comment colorier (mais donne un algorithme en $\Theta(n^2)$).
3. À noter que ce théorème porte sur les *cartes* (c.-à-d. les graphes *planaires*). La 4-coloration de graphes quelconques est dans NP.

La 3-coloration de graphe est dans NP : exemples de vérification

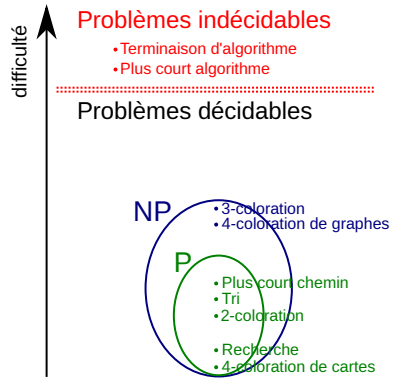
Nous ne connaissons pas d'algorithme en temps polynomial pour résoudre le problème de la 3-coloration de graphe planaire : le problème ne *semble* pas appartenir à *P*.

Néanmoins, le problème est dans *NP* : si la réponse est « oui », le certificat consiste simplement en la liste des nœuds, chacun avec sa couleur choisie. La vérification prend un temps linéaire :

- vérifier que tous les nœuds apparaissent dans la liste, que chacun a une seule couleur, et que chaque couleur est une des k couleurs autorisées ; et
- vérifier pour chaque arête que les deux nœuds qui sont ses sommets n'ont pas la même couleur.

La première partie prend un temps proportionnel au nombre de nœuds, la deuxième un temps proportionnel au nombre d'arêtes.

Résumé (d'une partie) de la classification des problèmes



Exemples de problèmes dans NP

Ouvrez à nouveau ce livre sur les algorithmes : vous trouverez des dizaines de pages de plus, décrivant des problèmes dans la classe NP—moins de pages que pour P, vu que ces problèmes seront seulement décrits, mais qu'aucun algorithme efficace ne sera présenté.

- problèmes d'optimisation sur les graphes : couvertures, coloriages, circuits, etc.
- problèmes sur séquences (texte ou ADN)
- satisfaire une formule logique, équivalence de formules
- problèmes de partitions en sous-ensembles (rendre la monnaie, affectations diverses, etc.)
- problèmes de regroupement
- problèmes d'ordonnancement
- problèmes de positionnement
- problèmes en mathématiques, physique, biologie, etc.

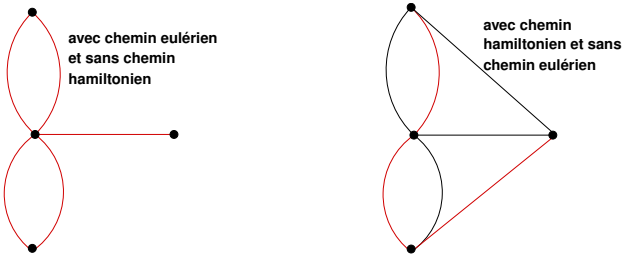
Presque tous les problèmes d'optimisation d'importance en industrie sont dans cette classe—et bien trop peu d'entre eux se trouvent dans P !

De nouveau des problèmes de chemin

Soit un certain nombre de villes reliées par un certain nombre de routes.

Question 1 : existe-t-il un itinéraire permettant de passer par toutes les villes en n'empruntant chaque route qu'une seule fois (**chemin eulérien**) ?

Question 2 : existe-t-il un itinéraire permettant de passer par toutes les villes en ne visitant chaque ville qu'une seule fois (**chemin hamiltonien**) ?



De nouveau des problèmes de chemin (2)

A priori les deux se ressemblent beaucoup et dans les deux cas on sait proposer un algorithme inefficace (recherche exhaustive) :

1. Générer tous les chemins.
2. Retenir le premier chemin répondant aux critères requis

mais...

☞ complexité **exponentielle** ! !

Sait-on faire mieux ?

De nouveau des problèmes de chemin (3)

La réponse est :

oui (on sait faire mieux) pour la question 1

☞ méthode d'Euler : complexité polynomiale

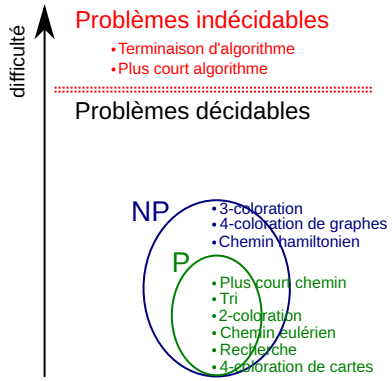
Ce problème est dans P

...et **non** (on ne sait pas faire mieux à ce jour) pour la question 2

☞ les meilleures méthodes à ce jour ne font pas beaucoup mieux que la recherche exhaustive : complexité exponentielle !

(la recherche de chemin hamiltoninen est dans NP)

Résumé (d'une partie) de la classification des problèmes



Et Prem, où est-il ?

Rappel : Prem(n) :

entrée : n entier naturel (≥ 2)

question : n est-il premier ?

Une solution proposée :

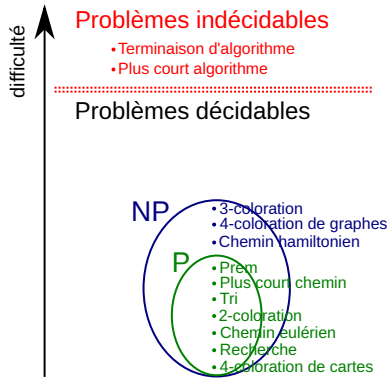
1. si n est pair, il n'est premier que s'il vaut 2
2. sinon, tester la division de n par tous les entiers impairs plus petits que $\sqrt{n}$

Cet algorithme est... ...*exponentiel !!*

☞ **Attention !** à la **taille** de l'entrée !!!

En 2002, Agrawal, Kayal et Saxena (AKS) donnent un algorithme en $\widetilde{\mathcal{O}}(\log(n)^{12})$, prouvant ainsi que **Prem est dans P**.

Résumé (d'une partie) de la classification des problèmes



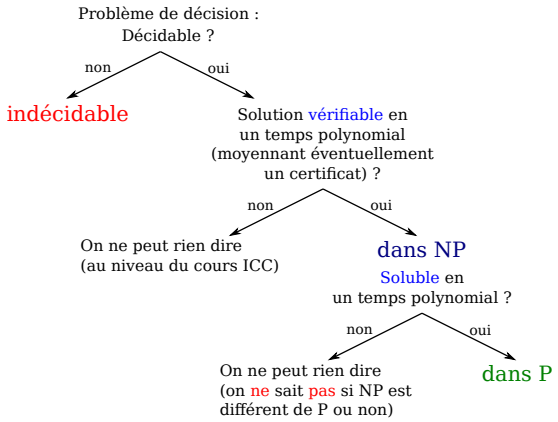
Les problèmes NP en pratique

La plupart des problèmes d'optimisation, de vérification, et de prédiction sont dans *NP*, mais apparemment pas dans *P*. Pourtant il faut tout de même bien produire des algorithmes et des réponses (en un temps raisonnable).

Alors que faire ?

- solutions exactes qui ne marchent que pour des cas particuliers
- solutions inexactes, mais avec certaines garanties
  par exemple, le débit maximum sera au pire 20% de moins que la valeur optimale
- solutions heuristiques, souvent excellentes, mais sans garanties
  par exemple, solutions dites « gloutonnes »
- solutions probabilistes, à l'aide d'un générateur de nombres aléatoires, qui donnent des garanties en moyenne

Les problèmes en pratique



Points à retenir

- Il existe plusieurs définitions formelles équivalentes de la notion de calcul, parmi lesquelles les machines de Turing.
- Il existe des problèmes apparemment simples qui ne peuvent pas être résolus par le calcul.
- Les limites absolues sont toutes liées à la notion d'autoréférence, qui est aussi un des outils indispensables de l'informatique.
- Les limites pratiques sont liées à la complexité du calcul.
- Il faut distinguer le calcul de la solution et la vérification d'une solution (tous deux sont nécessaires) :
  - P est la classe des problèmes dont on peut *calculer* une solution en un temps polynomial (en fonction de la taille de l'entrée)
  - NP est la classe des problèmes dont on peut *vérifier* une solution en un temps polynomial